Imprecise Results: Utilizing Partial Computations in Real–Time Systems

Kwei–Jay Lin
Swaminathan Natarajan
Jane W.–S. Liu

Department of Computer Science
University of Illinois at Urbana–Champaign
1304 W. Springfield Avenue
Urbana, Illinois 61801

{klin, swami, janeliu}@a.cs.uiuc.edu
(217) 333–1424

April 15, 1987

This work was partially supported by the NASA Contract NAG–1–613.

Imprecise Results:
Utilizing Partial Computations in Real–Time Systems

Abstract

In real–time systems, a computation may not have time to complete its execution because of deadline requirements. In such cases, no result except the approximate results produced by the computation up to that point will be available. It is desirable to utilize these imprecise results if possible. We propose two approaches to enable computations to return imprecise results when executions cannot be completed normally. The milestone approach records results periodically, and if a deadline is reached, returns the last recorded result. The sieve approach demarcates sections of code which can be skipped if the time available is insufficient. By using these approaches, the system is able to produce imprecise results when deadlines are reached. We describe the design of the Concord project which supports imprecise computations using these techniques. We also present a general model of imprecise computations which takes into account the influence of the environment, and show where our approach fits into this model.

1. Introduction

In real-time computations, deadlines are crucial. A hard real-time system must finish its computation before its deadline or something undesirable might happen. A real-time system typically consists of many small jobs. Normally, real-time systems rely on their schedulers to decide when and which job is to be executed next. Schedulers, in turn, rely on the knowledge supplied by the programmer and the specification about job execution times, job dependency relationships, and the system deadline to make the scheduling decision. This works well as long as the system environment is static in the sense that the execution time of each job is fixed, the deadline requirement is fixed, and the number of jobs to be executed is constant. Much research [Liu73,Blazewics83] has been done in using different policies to provide a static scheduling.

However, the scheme may not work if the environment is dynamic. For example, in a radar tracking system, the number of objects to be monitored may be dynamically changing all the time. In an environment where hardware may fail independently, a processor's load may change if it must take over some jobs which used to be performed by a now failed processor. If a job is in progress and suddenly the scheduler decides to change its deadline, it may have to stop its computation immediately. Thus, in the radar tracking application, if the screen is to be updated every 10 seconds, the job to determine the positions of all "visible" objects must complete within that time. If there are 100 objects in the field of view, but the procedure can track the exact locations for only 50 of them in the allotted 10 seconds, the system may miss some crucial objects among the 50 unprocessed objects. In this case, it is more desirable to obtain the approximate locations for all 100 objects. Such a system has two other advantages. It is flexible - if the sky suddenly gets cluttered, it should be possible to decrease the amount of time spent on each object, and thus get a more fuzzy view, but continue tracking all of them. Secondly, it lends itself to priority schemes - if, after viewing all objects, the system decides that some of these objects are more interesting, more time may be spent on them to get a more accurate positioning. Thus, sometimes it is advantageous in a real-time system not to carry out every job to completion, but to obtain results while computations are still incomplete.

In this paper, we propose several schemes for obtaining and utilizing results from partially completed computations. Since the computations did not complete, their results are *imprecise*. We suggest two different approaches that the programmer may use to construct procedures which are capable of producing imprecise results. The first approach is termed the *milestone* approach. Its central idea is to record meaningful partial results obtained at different points in the execution of the procedure; in the event of a deadline, the last recorded values are used as the result from the procedure. The second approach, called the *sieve* approach, defines sections of code that cause existing results to become more precise. For example, in image processing, enhancement routines are used which increase the contrast in the picture. The system is allowed to decide if sufficient time is available to perform extra enhancements. Not executing them amounts to producing a result of inferior quality but will require less processing time.

Both these approaches are particularly well-suited to iterative algorithms, though they can be applied to other kinds of procedures also. Many numerical algorithms involve iterating to improve precision [Pizer83]. Several studies [Basu80, Finance85, Turski84] have been conducted on how computations can be recast into an iterative form. The SIFT system [Wensley78] records partial results at the end of each iteration, and uses them for error checking in a replicated system. We think that using iteration to improve results is common enough in real-time systems for our techniques to be of widespread use.

The rest of this paper is organized as follows. Section 2 outlines the milestone approach and the sieve approach, and compares them. Section 3 presents a generalized model of imprecise computation, and the relationship of the two approaches to that general model. Section 4 describes Concord, a system that supports imprecise computations. Section 5 discusses extensions to the imprecise results technique. We give our conclusions in Section 6.

1.1. Relationship to other work

Both sieves and milestones are presented as language primitives that take advantage of system support to provide clean, easy-to-use aids for real-time programming. While it is possible to achieve the same effects with the constructs currently available in real-time languages, it would be awkward to do so, and the resulting program would not be semantically

clean or clear. So these techniques do address a problem which existing languages do not handle very well – that of terminating execution of a procedure before it is complete, and obtaining results from it. In fact, we feel that imprecise computation is a more general model than the traditional precise model. Always executing a procedure through to completion is a special case of imprecise computations in which the imprecision is reduced to a minimum.

Our approach to real-time system programming is different from other work on real-time programming languages such as Real-Time Euclid [Kligerman86], Ada[1] [DoD83] and Modula [Wirth77]. Most of them provide primitives so that programmers may have more precise control, and environments cause less variance, in execution time. They also provide ways to specify real-time delays in the code. For example, Real-time Euclid allows no dynamic data structures and recursive invocations which may have unpredictable execution time. Ada provides *delay* statements and *pragmas* to specify time constraints. Modula restricts context switches, and expects good programming technique and testing to ensure that deadlines are met. We consider these methods to be low-level, in that they expect the programmers to manage the timing issues themselves, with little or no assistance from the language or the system. The programmer has to explicitly ensure that each piece of code meets its deadline, and is only assured of minimal interference from the system. Confidence that the system will meet its deadline usually cannot be established until the program is tested under many dynamic circumstances.

Language primitives allowing programmers to specify only a desired completion time are not good enough for hard real-time systems. Our approach, instead, attempts to transfer the burden of meeting deadlines to the supporting system to a large extent, and only requires the programmer to assist it in some simple ways. Our primitives work at the procedure level, and leave it to the system to decide whether there is enough time to execute segments of code. Because they relate to the algorithm, our primitives can be added to any programming language. It is not our intention to suggest that this approach is in all respects superior to others; it requires a powerful scheduler, and in some cases may involve extra overhead. However, our

[1]Ada is a registered trademark of the U.S. Govt. – Ada Joint Program Office.

scheme can handle cases that traditional approaches cannot. It is our objective to explore an alternative approach to real–time programming and to demonstrate that it can be easily used.

2. Primitives for imprecise computation

2.1. The milestone approach

Many computations perform their operation in distinct stages or phases [Chandy83, Lampson76]. After each stage, the system has solved more of the problem and therefore produces an intermediate result which is closer to its final form. It is this idea which motivates our milestone approach of saving imprecise results of distinct significance. In case that the system is to terminate immediately, the last saved imprecise result is returned as the final result of the job.

In this approach, the primary assumption is that the result from the procedure increases monotonically in correctness as the procedure progresses towards completion. Therefore, the longer a procedure executes, the more correct the result is. This corresponds to a continuous correctness function as shown in Figure 1. $C_R(t)$ is the correctness of the result at time t. The time t_m represents the minimum time it takes for the procedure to begin to produce a useful result. In practice, because software consists of discrete chunks of code (e.g. loops), correctness can be better represented as a staircase function as shown in Figure 2. Each step indicates the completion of another phase of the computation, such as another iteration of a loop. We suggest that each step could form a "milestone", and the intermediate results at the end of each step could be saved. If the procedure terminates prematurely, the last set of recorded values could be returned as the most accurate results available from the procedure.

2.1.1. Features of the approach

The system model for the milestone approach is that of client/server. The client, or caller procedure, invokes the server, or callee, providing it with a set of input parameters. The deadline of the requested operation can be part of the input parameters if it is known to the client at the moment of invocation. Otherwise, the system needs to provide some mechanism to

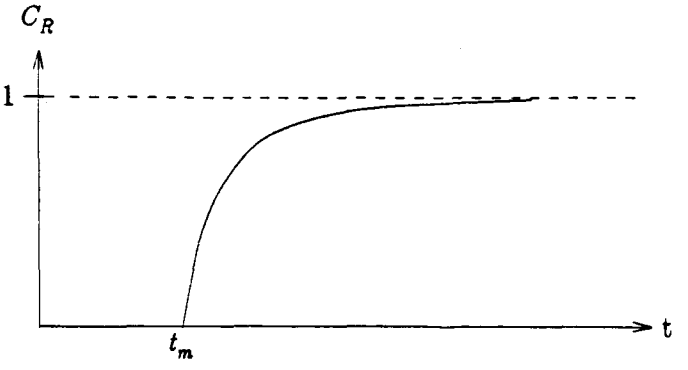

Figure 1. A continuous correctness function

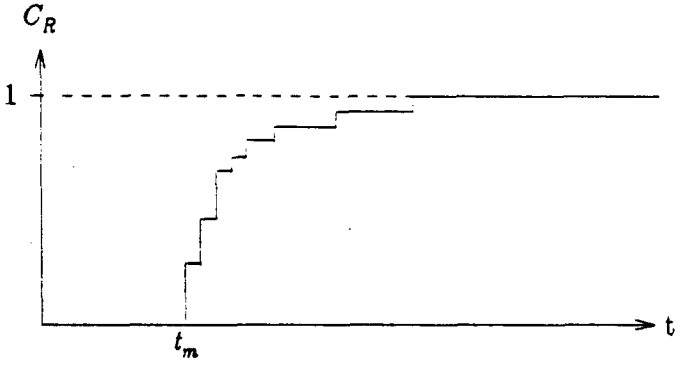

Figure 2. A staircase correctness function

inform and to stop the server when the deadline becomes known. The callee starts executing the procedure when the request is received. If enough time is available, it produces the required results and returns them to the caller. On the other hand, if the deadline is reached while the callee is still executing, it terminates immediately, and returns a set of imprecise results instead.

Two language primitives are used by the programmer to specify a computation involving milestones. Our proposed primitives use the syntax of the C language. In the callee. the *impreturn(variable-list)* command is used to indicate that a milestone has been reached. The current result value is always the first variable in the list. The values of the variables in the list other than the result variable constitute the intermediate results available at the milestone. This list should include any global variables that may be modified by the procedure, because if the

global is modified subsequent to the milestone, but computation is terminated before the next milestone, the global may end up with a value that is inconsistent with the rest of the system. This variable-list would thus, in the general case, be a superset of the values normally returned as results if the call ran through to completion. In addition to global variables, the variable-list may also include some error indicators which may be used by the caller to determine how precise the result is.

The caller uses the *impresult(procname, handler)* command to indicate whether milestones are to be recorded by the procedure named *procname*. It is possible that even if the callee procedure contains specifications of milestones, the caller may not wish to record milestones and receive partial results. This may be because there is enough time to complete the request, thus there is no need to incur the additional overhead, or because imprecise results are unacceptable. The impresult command also specifies a handler which interprets the values of the intermediate results returned along with the result variable. The handler accepts the complete set of intermediate values, and performs appropriate modifications to the result variables so that they can be meaningfully utilized. For example, a caller may contain statements like

```
...
impresult(sum, extrapolate);
total = sum(arrayA);
...
```

and the handler *extrapolate* is defined as

```
extrapolate(z, x, y) float z, x, y;
{
        z = z * y / x;
        return(z);
}
```

The impresult statement associates the handler *extrapolate* with the server *sum* and triggers the milestone primitives implemented in *sum*. Assuming that the impreturn statement in *sum* returns the total array size in the third variable (i.e. y) and the number of array elements already processed in the second variable (i.e. x), the handler can compute the expected result for the whole array using the current result z by the formula z*y/x and returns that as the precise result. In general, the handler may perform the function of correlating the variables in the list, applying any transformations or compensation necessary, and arriving at a set of modifications to the intermediate result. The handler is invoked only if the computation returns an imprecise result. If the callee terminates normally, the handler is bypassed entirely.

It is necessary to specify an imprecise result handler before invoking a server for the first time, to trigger the milestone mechanism. By default, the system assumes that the caller expects a precise result. It is also possible that the caller may want to associate different handlers within different calling instances so that imprecise results can be interpreted in different ways. A program can change the association between a callee and its handler by assigning a different handler using the impresult statement. There are two special handlers defined for the impresult command: a NULL handler indicates that the milestone mechanism in the callee should be utilized, but no special handling of the imprecise result is necessary (i.e. the imprecise result is accepted as is); an OFF handler indicates that the milestone mechanism is to be turned off. This is needed because once a handler has been associated with a procedure, it continues to be associated until the next invocation of impresult.

2.1.2. Design and semantics of milestone primitives

One of our design decisions for both the milestone primitives and the sieve primitive to be discussed later is that it must follow the C syntax so that it is readily compilable using a standard C compiler. There are many advantages to this approach besides saving the trouble of changing the C compiler. First, an application program designed with the imprecise system support in mind can be run on a normal system without modification. All the normal system needs to do is provide two no–op functions for the impresult and impreturn statements in order to make it behave like normal application programs. Second, it provides transparency. It is

possible to change the mechanism for producing imprecise results without changing the source and recompiling. It is even possible to provide special architectural support for the primitives without rewriting any application.

The semantics of the primitives is similar to that of exception handling [Goodenough75]. Recording an imprecise result with impresult is like raising an exception condition; the difference is that the callee is terminated in the latter case of exceptions but not in the former case. Executing an impreturn is like handling an exception, especially if the exception model is that of resumption. The impresult mechanism resembles the signal mechanism of C. The milestone approach itself is a special case of exception mechanisms. The reason for the imprecise "exception" is the insufficient execution time.

2.2. The sieve approach

The sieve approach is based on the concept of *sieve* functions. Sieves are functions whose only purpose is to "refine" their inputs, i.e. make them more precise. The outputs correspond in number and type to the inputs, and are semantically more precise versions of them. An example of this is the Gauss–Seidel iterations method for solving simultaneous equations [Stark70]. In this method, all the unknowns are initialized to arbitrary values, usually 1. Then each equation is used to solve for one unknown, till all the unknowns have a new set of values. Thus, a complete iteration of this technique starts with a set of values, and refines them to produce better approximations to the value of each unknown.

Sieve functions have the delightful property that it should not be fatal to the correctness of the procedure if a sieve is not executed. Such procedures are more common than it might seem. For instance, the loop body of many iterative functions can be viewed as a sieve because each iteration computes a closer approximation to the final answer. In practice, it is not possible to tell syntactically whether a given procedure is a sieve or not. Thus, in the above example of Gaussian iteration, the input values might be all 1.0, and the outputs might be 1.8, 2.4, 1.7 and so on. Therefore it is left to the programmer to ascertain that any procedure designated as a sieve really does refine its inputs. Actually, any section of code may be designated as a sieve if it refines the result variables and there is no other side effect produced by the code.

The sieve approach is based on the assumption that sieves perform computation in order to increase precision, and as a corollary, they may be bypassed (not executed) and the effect would be simply loss of precision. This provides a way to execute procedures faster at the cost of imprecision. The programmer specifies that some of the computations are sieves, and the system has the option of either executing the sieve or discarding it. This decision is made by the system according to the nearness of the deadline. If the deadline is very close, the system may opt to omit all sieves and just conduct the simplest computation through the rest of the procedure to return an imprecise result.

The semantics of the sieve approach can be understood in terms of the staircase correctness function. Each step of the correctness function corresponds to the execution of a sieve, which increases the correctness by an amount corresponding to the height of the step, and takes time corresponding to the width. Omitting a sieve has the effect of removing a step, resulting in the rest of the correctness function moving to the left (taking less time) and finishing at a lower level (the result is less precise). This is shown in Figure 3, where omitting a sieve reduces the correctness from 1 to C_1, while the execution time decreases from t_2 to t_1.

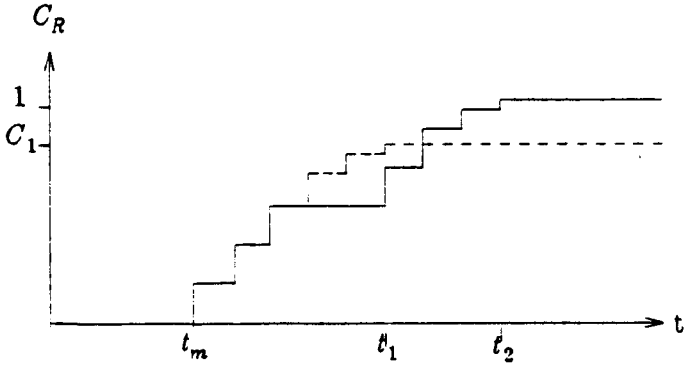

Figure 3. Omission of a sieve

2.2.1. Features of the approach

The sieve approach is concerned only with the callee. A section of code in the callee which constitutes a sieve is bracketed by the *sieve* command:

```
sieve {
        ....    /* code for the sieve */
        ....
}
```

During execution, the system executes these sieves if the deadline has not yet been reached. Once the deadline is reached or getting close, the sieves are skipped, and a result is produced as fast as possible. One significant aspect of this is that the result may be produced not at the deadline but shortly thereafter. This is acceptable if the deadline is flexible, i.e. it is a soft real-time system. It is not acceptable if the deadline represents an emergency, such as a robot crashing into a wall. However, if the program can be written so that the sieve is the last section of code in the procedure, skipping it makes the result immediately available. Another noteworthy aspect is that using the sieve approach, the caller has no way of telling whether the procedure executed normally or some of the sieves were skipped to make the deadline.

This approach is even more effective if the expected time of execution of each sieve is also provided. This gives the system more flexibility in scheduling to meet hard deadlines. The system can decide which sieves to execute so that the computation can be finished in time and yet the result is most precise. It can also make trade-offs, if some of the sieves are to be executed repeatedly. For example, if sieve S1 is normally iterated 100 times and sieve S2 is to be iterated 30 times, and if half the normal execution time is available, each sieve may be executed half as many times as requested. If the two sieves work on different variables, this has the effect of distributing the imprecision instead of lumping it all on one variable.

2.3. Comparison of the two approaches

The milestone approach is basically a pragmatic, system-level technique. It is more difficult to formalize, but easy to use from the programmer's point of view. It is explicit in permitting a variety of imprecise results from the procedure. The selection among these mechanisms is implicit and inflexible – it is chosen by reaching a deadline. It does not need complex support. The milestone approach is a passive approach in that termination is determined by the external intervention. The program behaves optimistically – it always assumes that it has enough time to finish. Saving imprecise results is simply a precaution.

The sieve approach is a semantic, language-level technique. It can be better formalized, using the formalism similar to a guarded command [Dijkstra75]. It is implicit in the way it specifies the possible set of imprecise results returned, but explicit in the selection mechanism, which is built into the system. The sieve approach is an active approach in that the program explicitly decides whether a sieve is to be executed or not. The program decision is always pessimistic; it executes a piece of code only when it knows that it can be finished.

It is expected that users will use sieves when the computation naturally takes the form of successive refinement of values, and use milestones for saving intermediate results after distinct stages of computation. Actually, users may use both primitives in one program. It is desirable to use milestones to save and produce some basic results before using sieves to improve them further as long as there are time and resources available.

3. Model of imprecise computations

Computations performed by a computer program are usually modeled as transform functions which transform some input values and an initial state of the program to a final state with some output values produced. In reality, however, a program execution with the same input and initial state may actually produce different results due to the differences in the environment. In some applications, these differences are expected, thus all different results are considered to be correct. In other applications, these different results may be regarded as approximations to a good result and therefore are accepted with reservations.

In a distributed system, variations in communication delays may lead to different execution sequences. If two separate requests are entered into an airline reservation system simultaneously from different places, either one may be processed first. The nondeterminism inherited in the system may cause different results to be produced, but both execution sequences are correct. In a real–time system, deadline constraints may cause a procedure execution to be terminated prematurely. In a fault–tolerant system, crashes and recoveries can result in various possible execution sequences, and the results obtained may not be identical for all of them (due to "fail–soft" mechanisms).

It is therefore desirable to have a model of computation that includes the factor of environments. The environment can be modeled as a state machine which interacts with the state machines of concurrent programs. However, such modeling is complicated and often unnecessary. It is often enough to introduce the environment as a parameter of a computation and assume that it is unchanged throughout the computation. For example, deadline is a parameter which can be specified before the computation starts. The fact whether it is predefined or not is irrelevant to the actual computation performed. Some aspects of the environmental variations can be hidden from the program altogether, they will not be shown in the computation model. Other aspects may affect the program execution and therefore should be dealt with at the program and user level.

Imprecise results caused by time restrictions can be viewed as a special case of a more general model which takes into account the effect of variations in environment on computations. In a real–time system, environment inputs are usually handled by the system scheduler which in turn changes the scheduling decision in the form of deadlines and priorities. The environment can also influence the computation directly by crashing a particular processor or by interrupting a computation.

We define a general parameter $\mathbf{E}$ to be the subset of the environmental state which will affect the execution of a program P. We model the computation C which is an instantiation of P by the following transition function:

$$C: \mathbf{I} \times \mathbf{S} \times \mathbf{E} \rightarrow \mathbf{O} \times \mathbf{S} \tag{1}$$

where **S** is the set of states of the program P, **I** is set of input values, and **O** is the set of output values. In other words, given an initial state S, input values I, and an environment state E, we have

$$(O, S') = C(I, S, E) \tag{2}$$

for some output value O and final state S'. We are not concerned here with any side effects the computation C might have on the environment. The fact that the result of computation C might lead the environment into a state E' different from its initial state E is thus not expressed explicitly in (1) and (2).

We can use the model to reason about the different behaviors of a program under different environments. A program which produces different results under different environments is logically equivalent to a set of programs free of environmental influences that produce corresponding results. Similarly, using a single server to produce a series of imprecise results is logically equivalent to executing many copies of the server with different execution times. Therefore, the original program does not uniquely define the actual computation which will be performed. Rather, it defines a set of possibilities.

The imprecise computation model can provide better performance and reliability because it can endure imprecisions in the environment and produce a set of results. The problem is how to choose the best result from the set of possible results. In a multi-version system, the result from a server running on the most precise hardware is usually regarded as the most accurate. Sometimes the result from the fastest server is selected if the performance is more important than accuracy.

In this paper, we are particularly interested in environmental imprecision in execution time and its effect on the output results. If the precision of the result produced increases monotonically with execution time, then we can always choose the result produced by the computation with the longest execution time. Most applications do possess this monotone property. Our primitives, based on the imprecise computation model, therefore provide users with a simple way to obtain better results from incomplete computations.

4. The Concord system

We are designing the Concord system [Lin87] which can support imprecise computations. The system provides both a programming tool so that a user can design imprecise computations, and run-time support so that imprecise results can be reliably and meaningfully returned. Such a system would give users much greater flexibility in implementing their applications.

The Concord project uses the milestone approach to permit procedures to return imprecise results. To record the milestone results, and return the latest if the deadline is reached, a *supervisor* is introduced between the caller and callee. The caller sends the call to the supervisor. The supervisor starts up the callee, and periodically receives the milestone results from it. The supervisor keeps track of the last set of results received. If the deadline is reached, it terminates the callee, and sends the imprecise result to the caller. If the callee completes, the value returned is passed on to the caller. The configuration is shown in Figure 4.

This implementation has the advantages of being simple, having comparatively little overhead, and of being modular. It separates the functions of the callee from the system support issue of recording the results. On a distributed system, it has the additional advantage of fault-tolerance if the supervisor is on a different machine from the callee.

The introduction of the supervisor is not the only way to implement the milestone approach. The system can reserve a milestone space in the callee's address space for

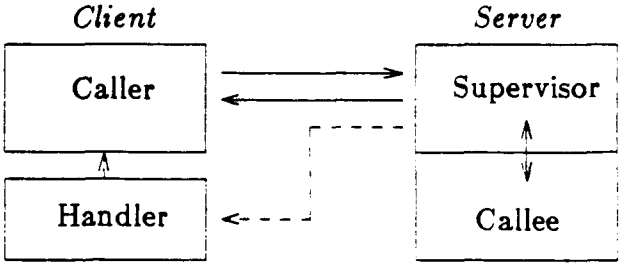

Figure 4. Concord process structure

intermediate results. The size of the milestone space is exactly the same as the size of the final result variables plus the total size of the variable list specified in the impreturn command. All these are known at the compile time. During run-time, when an impreturn statement is encountered, all the system has to do is to copy the current value of all those variables into the milestone space. Since it is always the last set of values that are returned, only one such set needs to be kept. When an imprecise value is returned, the handler is invoked with the intermediate results.

Concord also supports the sieve approach. As discussed later, Concord includes a sophisticated system scheduler. Each sieve primitive queries the scheduler to determine whether enough time is available for the sieve to be executed. Using the C preprocessor, the *sieve* primitive is translated to *if (enough_time())*, which queries the system scheduler. This preprocessor statement is contained in a header file which is part of the Concord language library, discussed in the next subsection.

4.1. Run-time system implementation

Concord will be implemented on a network of Sun workstations. As mentioned before, we uses C as the target language because it is popular and easy to work with. We may switch to some other language which is more suitable for distributed applications after we have some

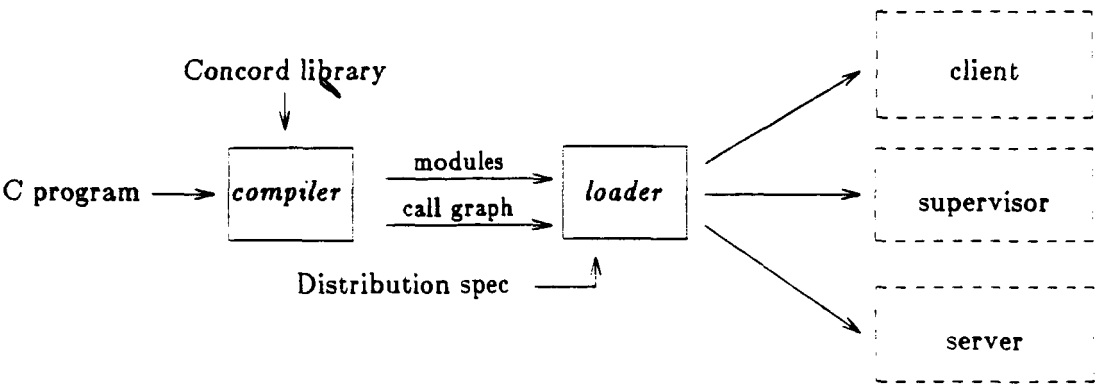

Figure 5. Concord language processor

experience with Concord. Concord has two components: the language processor and the run-time system. The language processor, shown in Figure 5, includes libraries for the impreturn and impresult primitives, and a loader for distributing the modules produced by the compiler among the different processors in the system. The sieve primitive is very easy to support. It is simply converted to a call to the system scheduler to determine if enough time is left to execute the sieve. The impreturn primitive is a procedure call to the supervisor, which records the arguments provided as a milestone. The impresult primitive is implemented as a procedure call to the run-time system, to specify whether the RPC messages should be issued to the supervisor, or directly to the callee in the event that milestones need not be recorded. The impresult procedure also associates a handler to the RPC, so that if the computation terminates prematurely, the run-time system will invoke the handler to process the imprecise result.

The Concord run-time support is a layered distributed system which can support imprecise result communications. For the Concord prototype we plan to reuse as much as possible the run-time system designed for the Resilient Procedure (RP) project [Lin86]. Both projects support imprecise computations. RP supports imprecision arising from multi-version software [Chen78] and hardware while Concord supports imprecision in execution time. It is our ultimate goal to unify the projects together to support general imprecise computations.

4.2. Scheduling in Concord

The proposed Concord system includes a sophisticated scheduler. This scheduler is designed to take advantage of the possibilities for returning imprecise results from partial computations. This has substantial impact on scheduling, because the emphasis changes from meeting deadlines to finding the optimum schedule that will minimize the imprecision in the results. The concept of *average error* of all results computed is defined, and the objective is to minimize this. We are currently considering various heuristic algorithms to produce near-optimal schedules [Liu87]. The scheduler also addresses the problem of not allowing imprecision in a particular value to accumulate too much due to repeated calls to some procedure.

Sieves interact with this scheduler directly. Each sieve command is translated into a call to a system scheduler routine which makes the decision to ascertain whether sufficient time is left

before the deadline to execute the sieve. If all sieve functions include time estimates, the scheduler can analyze all these estimates at the beginning of the procedure, and determine which sieves are to be executed in order to optimize the use of the available time, similar to static scheduling.

To use this approach, the scheduler must be intelligent in order to get the proper benefit from the scheme. In general, this is not a problem because most real-time systems have a sophisticated scheduler. The possibility of skipping certain sieves makes the scheduling decision much more flexible.

5. Extensions to the scheme

5.1. Applications in other areas

Imprecise computations as a technique for handling environmental variations has applications in fault tolerance and distributed systems. The milestone technique can be used to provide checkpoints for fault-tolerance. If the supervisor is on a machine different from the callee, and the callee's machine crashes, the supervisor has a partial result which it can return to the caller's handler. The handler has the option of either accepting the imprecise result, or resuming the callee from that checkpoint on a different processor. It can even invoke a different procedure to perform some patchwork to make the result more acceptably precise.

Since the model of computation includes the idea of a procedure being actually a set of computations among which one is selected by the environment, it can easily be extended to include the idea of N-version programming [Chen78] and Resilient Procedures [Lin86] for fault tolerance. In both techniques, several computations are performed and the final result is chosen from among the different results obtained.

In a distributed system, there are often variations in processor precision, memory etc. among different nodes. The model of imprecise computation can be used to specify what environment is required for each computation, facilitating optimal resource allocation. Also, if some resource is not available, or is deficient in some respect (such as less precision) it is possible to use the imprecise result to gauge if the error is within acceptable limits - the handler will

realize that an environmental variation has occurred, and can take appropriate action. It is also possible to simulate some resource which is not available, to bring the precision back to acceptable limits.

5.2. Future extensions

The multi-version concept can be used to define several versions of a procedure, of which one will be executed for which the environment is appropriate. Thus, it may be possible to have several versions of a procedure, which take differing amounts of time, and one will be picked which best matches the time available. Since the selection process is entirely a matter of matching available environmental resources to requirements, it can be done entirely at the system support level.

The supervisor in the milestone approach can take a more active role by using extrapolation to predict the final result instead of merely returning the last one. This is possible when the successive milestones form a pattern, as would be the case if they were the results of successive iterations of a numerical algorithm. The extrapolation procedure can be user-defined.

Computations which do not increase monotonically in correctness can be made suitable for the milestone approach by appropriate choice of milestones. The idea is to pick points on the curve of fluctuating correctness, so that they form a monotonically rising curve. This will satisfy the criterion that later results should be more precise than earlier ones.

The results recorded in the milestone approach can be enhanced to constitute a checkpoint so that it is possible to perform forward recovery for fault-tolerance purposes. If the values constitute a checkpoint, and the partial results are not acceptable, they can be used to resume the computation from where it was terminated.

6. Conclusions

The basic concept of returning imprecise results from procedures to allow for environmental variations has several applications, particularly in the area of real-time programming. It can be used to allow the user to separate out the algorithm code from the code that deals with

massaging results to allow for machine- and environment- dependent limitations, and abstract out the non-algorithmic code in special handlers. On the one hand it allows the system to take up as much of the burden as possible in the form of system support, and on the other hand it gives the user power, flexibility and control. The approaches discussed are not difficult to implement, and do not involve much overhead.

We see our approach as being the "natural" way to handle partially complete computations. When humans are faced with the same situation of not being able to complete a task, they tend to store the results obtained so far, because those results may be useful later. It seems reasonable for computers to do the same, rather then discard incomplete computations. We would like to see the concept of imprecision to be built into programming languages intended for production code.

References

[Basu80]

Basu, A.K., "On development of iterative programs from function specifications," *IEEE Trans. Software Eng.*, vol. SE–6, pp. 170–182, Mar. 1980.

[Blazewics83]

Blazewics, J., J. K. Lenstra, and A. H. G. Rinnooy Kan, "Scheduling subject to resource constraints: Classification and complexity," *Disc. Applied Math.*, vol. 5, pp. 11–24, 1983.

[Chandy83]

Chandy, K. M., J. Misra, and L.M. Haas, "Distributed deadlock detection," *ACM Transactions on Computer Systems,* vol.1, No.2, pp. 144–156, May 1983.

[Chen78]

Chen, L. and Avizienis, A., "N–version programming: A fault–tolerance approach to reliability of software operation," in *Proc. 8th Symp. Fault–Tolerance Computing,* 1978, pp. 3–9.

[Dijkstra75]

Dijkstra, E.W., "Guarded commands, nondeterminacy, and formal derivation of programs," *Comm. ACM,* Vol. 18, pp. 453–457, Aug. 1975.

[DoD83]

Reference Manual for the Ada programming language, (ANSI/MIL-STD-1815A), U.S. Dep. Defense, Washington D.C., Jan. 1983.

[Finance85]

Finance, J.-P., and Souquieres J., "A method and a language for constructing iterative programs," *Science of Computer Programming,* vol.5, pp. 201-218, 1985.

[Goodenough75]

Goodenough, J.B., "Exception handling: issues and a proposed notation," *CACM,* vol. 18, pp. 683-696, Dec. 1975.

[Kligerman86]

Kligerman, E., and A. D. Stoyenko, "Real-time Euclid: a language for reliable real-time systems," *IEEE Trans. on Software Eng.,* vol. SE-12, No. 9, pp. 941-949, Sep. 1986.

[Lampson76]

Lampson, B.W., and H.E. Sturgis, "Crash recovery in a distributed storage system." unpublished paper, Comp. Sci. Lab., Xeros Palo Alto Research Center, Palo Alto, CA. 1976.

[Lin86]

Lin, K.-J., "Resilient procedures - an approach to highly available system," in *Proc. IEEE Computer Society International Conference on Computer Languages,* Miami, pp. 98-106. Oct. 1986.

[Lin87]

Lin, K. J., S. Natarajan, and J. W. S. Liu, "Concord: A distributed system making use of imprecise results," Technical Report No. UIUCDCS-R-87-1330, Department of Computer Science, University of Illinois at Urbana-Champaign, 1987.

[Liu73]

Liu, C. L. and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. Assoc. Comput. Mach.,* vol. 20, pp. 46-61, 1973.

[Liu87]

Liu, J. W. S., Lin, K. J., and S. Natarajan, "Scheduling real-time, periodic jobs using imprecise results," submitted for publication.

[Pizer83]

Pizer, S.M., *To Compute Numerically,* Little, Brown and Co., Boston, MA, 1983.

[Stark70]

Stark, P.A., *Introduction to Numerical Methods,* The Macmillan Company, London, 1970.

[Turski84]

Turski, W.M., "On programming by iterations," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 175–178, Mar. 1984.

[Wensley78]

Wensley, J.H. et al, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE,* vol.66, pp. 1240–1255, Oct. 1978.

[Wirth77]

Wirth, N., "Towards a discipline of real-time programming," *CACM,* vol. 20, pp.577–583, Aug. 1977.